CSCE 313-200
Introduction to Computer Systems
Spring 2025

Memory

Dmitri Loguinov
Texas A&M University

April 10, 2025

Chapter 7: Roadmap

7.1 Requirements

7.2 Partitioning

7.3 Paging

7.4 Segmentation

7.5 Security

Part III

Chapter 7: Memory
Chapter 8: Virtual RAM

Requirements

Main memory services of the OS:

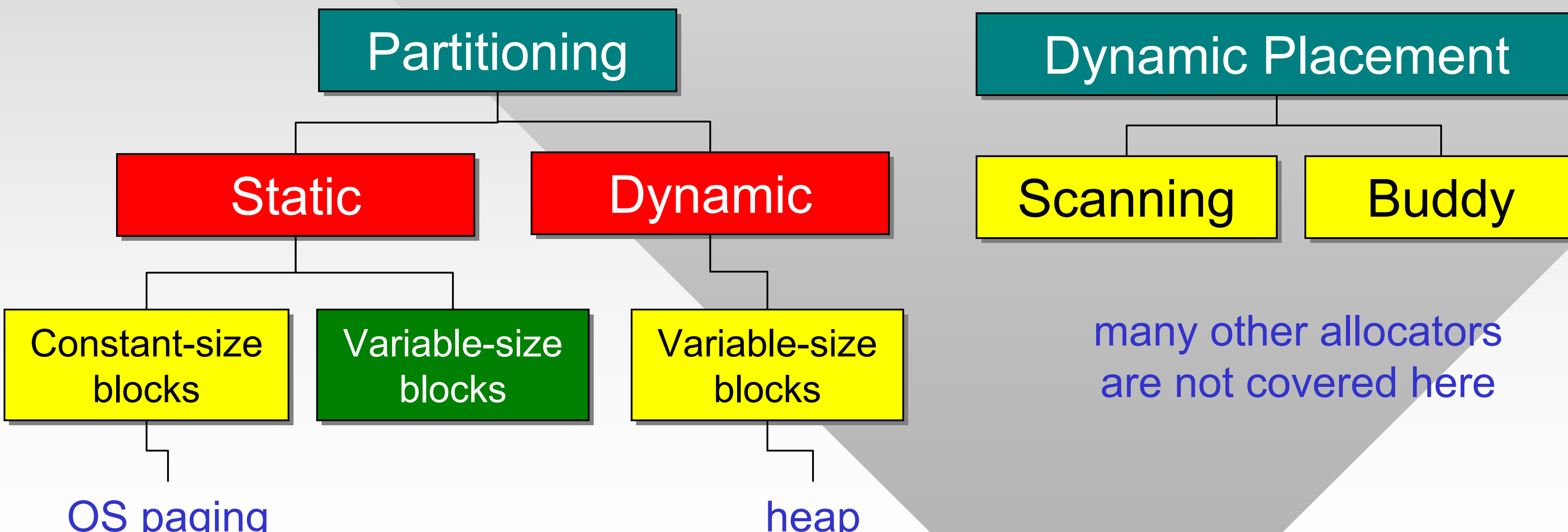
- 1) Dynamic allocation/deletion
- 2) Process & data relocation
  - Transparent fragmentation of process data/code within RAM and swapping to disk as needed
- 3) Protection
  - No unauthorized access to space of other processes
- 4) Sharing
  - Ability to map portions of RAM between different processes

Memory manager, address virtualization, hardware support

Chapter 7: Roadmap

Memory Management

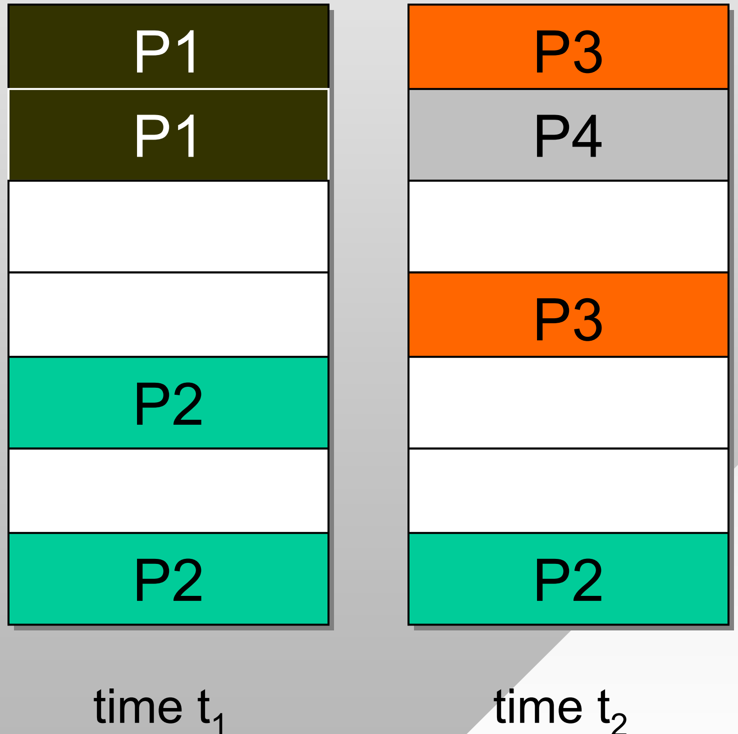
- Memory allocation is a complex problem
  - We examine only the most basic approaches
- Partitioning: type of RAM segmentation into blocks
- Placement: actual block allocation algorithms



Note: memory heaps have nothing to do with priority queues

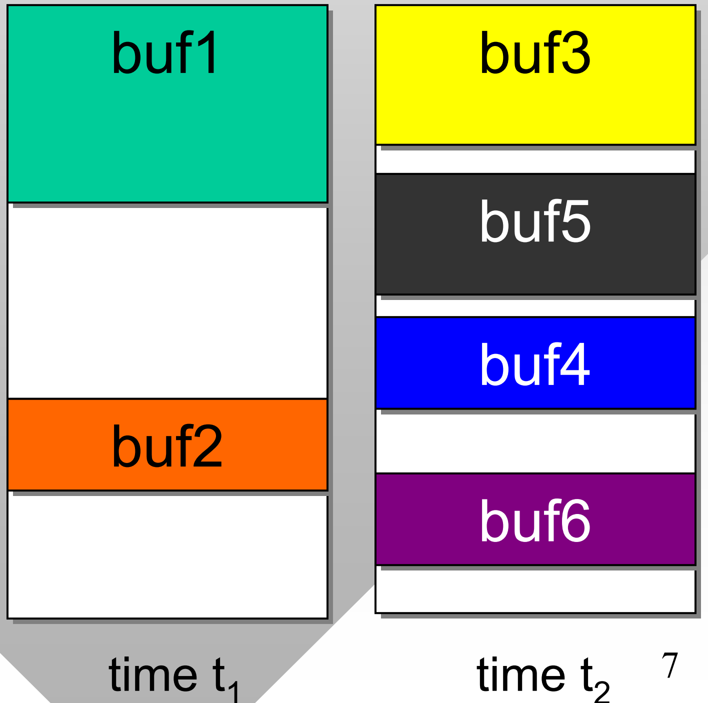
OS Partitioning

- Static partitioning defines block boundaries a-priori
  - Process may hold any number of blocks, which may appear to it as contiguous space
  - Mapping done in hardware
- Suffers from internal fragmentation
- Blocks may be of constant or variable size
  - For simplicity, most kernels have constant-size blocks called pages
- Each page must be a power of 2 (usually 4 KB)



Heap Partitioning

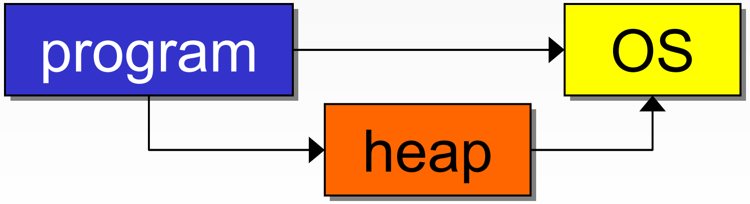
- Tweaking virtual-page tables is slow and a privileged operation; allocation rounded to nearest page size
- Idea: add memory management to user space that can satisfy small buffer requests with less overhead
- Dynamic partitioning (heap) grabs pages from the OS, then splits them into smaller chunks in user space
  - Much faster, but leads to external fragmentation
- More difficult to manage due to variable-size blocks



Heap Allocation

```
void f (void) {
    int a;          // on the stack
    // ptr on the stack, buffer on the heap
    char *buf = new char [100];
    // ptr on the stack, buffer from the kernel
    char *OSbuf = VirtualAlloc (...);
}
```

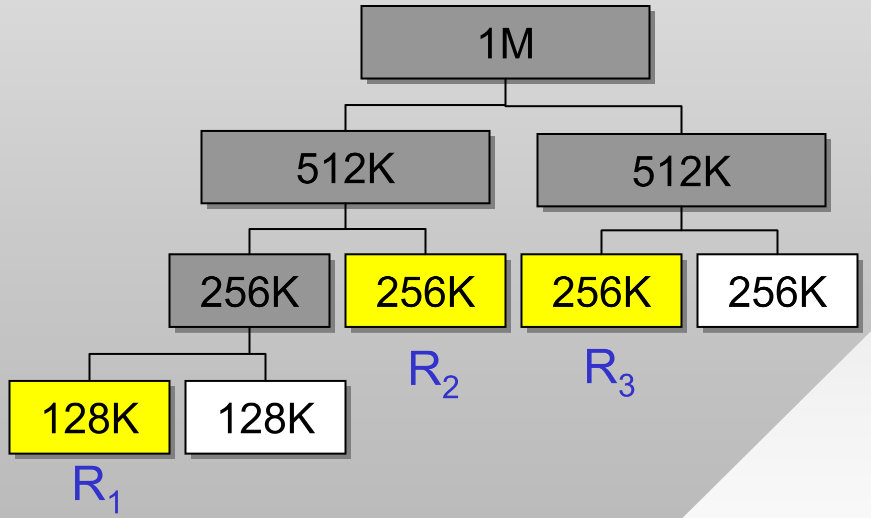
- Memory is typically allocated from:
  - Stack (local variables)
  - Heap (new/malloc)
  - OS (VirtualAlloc)
- We are now concerned with heap
  - OS issues covered during next class



- Scanning
  - Linearly search through RAM (or list of empty blocks) to find empty blocks to allocate
- Search types:
  - First fit: scans from start
  - Best fit: finds the smallest free block that satisfies the request
  - Next fit: searches from the last allocation forward
- E.g., Unix SLOB allocator for simple (embedded) devices

Heap Allocation

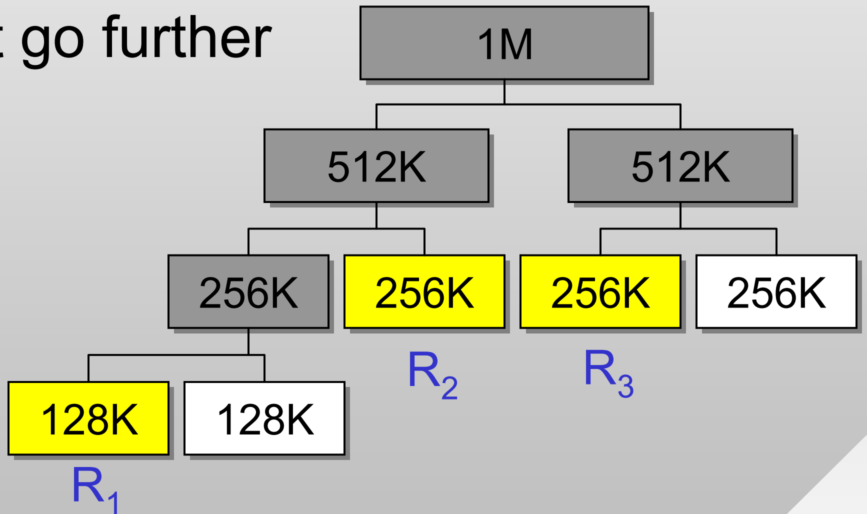
- Buddy System
  - Organizes OS chunk into blocks that are powers of 2
  - Smallest block has size 2^L, largest 2^U
- Request of size R arrives
  - Find a block with size that's nearest power of 2
  - If no such block exists, split larger free blocks in half until a block of correct size is available



- Example: U = 20, L = 12
  - First request is R_1 = 90K
  - Then requests R_2 = 150K, R_3 = 200K arrive in that order

Heap Allocation

- To free a block, check if the matching buddy is free
  - If so, combine and free the larger block
  - Process repeats until we can't go further
- Example:
  - Release order: R2, R1, R3
  - Which nodes are combined?



- Method drawbacks?
  - Both internal and external fragmentation, constant splitting & merging
- How to implement this scheme efficiently?
  - First problem is finding free blocks in U-L time
  - Second problem is merging buddies in U-L time

Heap Allocation

- Given R, first determine the size of target block
  - Needs to be the nearest power of 2 above or equal to R
  - Use _BitScanReverse to get the highest bit set in DWORD
- Free blocks are kept in queues, one for each level
  - Try popping a block from the needed level, if nothing there, go hunting for a larger block up the tree

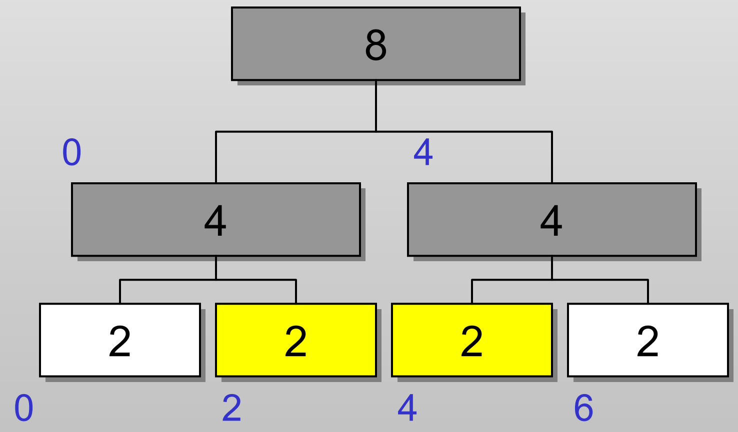
```
int levels = U – L + 1;
// queue of free blocks
Queue *fb = new Queue [levels];
char* Alloc (int R) {
    if (R == 0)
          return NULL;
    // index of the queue in [0, levels-1]
    DWORD qIdx = GetIndex (R);
    // search for the nearest empty block
    int i = qIdx;
    while (i >= 0 && fb[i].size() == 0)
          i--;
    // anything available?
    if (i < 0) return NULL;
```

```
    // if so, split them down
    for ( ; i < qIdx; i++) {
          ptr = fb[i].pop();
          fb[i+1].push (ptr);
          fb[i+1].push (ptr + 2^(U-(i+1)));
    }
    // pop our block
    ptr = fb[qIdx].pop();
    return ptr;
}
```

- Block with index i has size 2^{U-i}

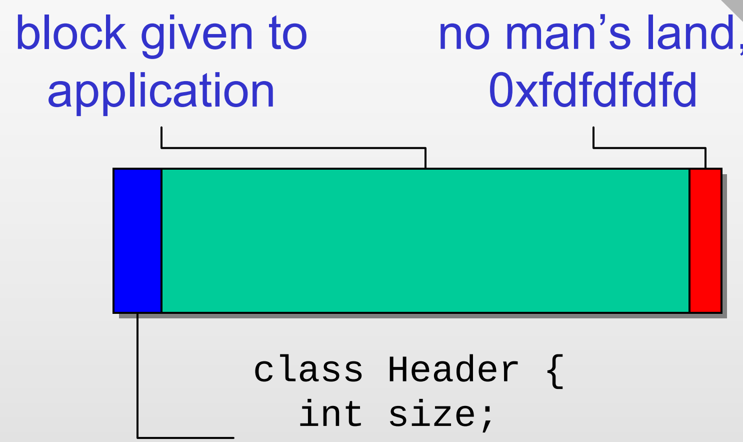
Heap Allocation

- How to free blocks and find who their buddies are?
  - Assume both ptr to start of block and its size are known
- XOR block ptr with its size
  - This gives a ptr to buddy block
- One approach is to scan the queue of free blocks, if buddy is there, merge



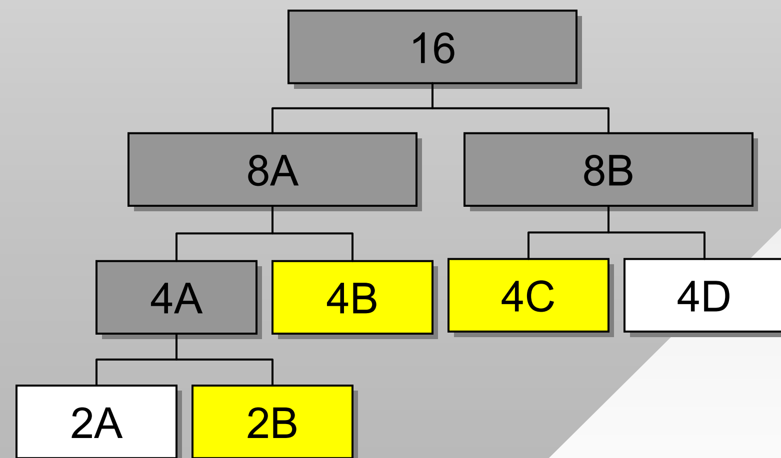
- However, this requires more overhead than we wanted (i.e., 2^{U-L+1} worst case)
- <u>Idea:</u> store allocation state with the blocks
  - Reserve a shadow buffer at the start of block

Heap Allocation




```
class Header {
  int size;
  bool free;
}
```

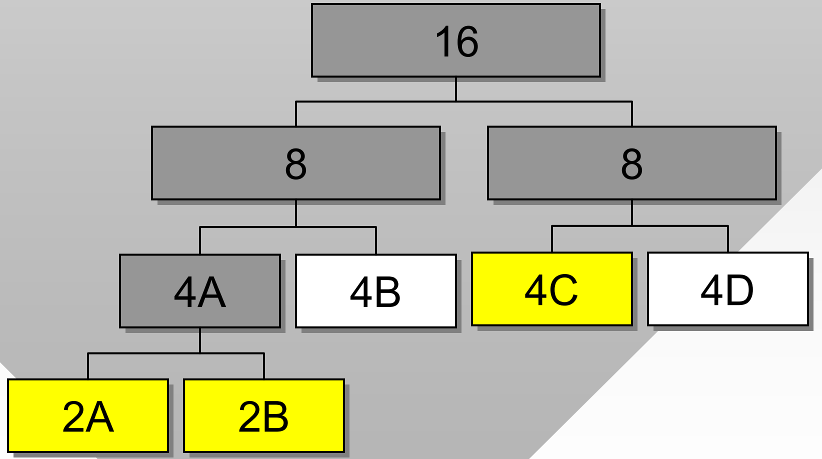
- Merge happens only when our buddy is free and their size matches ours
- Example when checking only the free flag is insufficient?
  - In this tree, 4B when freed will attempt to merge with 2A since starting address of 2A and 4A is the same (i.e., 0)



- To expedite efficient removal from queues, block headers may be organized into a doubly linked list instead of using separate queues

Allocation

- Modern malloc (stdlib, glibc) are variations on buddy
- Unix *slab* allocator
  - Do not merge up when expecting new requests of similar size and maintain a cache of small blocks
  - Threshold size for merging may be guesstimated from prior request patterns or hardwired ahead of time
- Low fragmentation heaps
  - When multiple options are possible, attempt to optimize continuity of space
  - 4B might be preferred over 4D for new splits



- Per-CPU heaps with better concurrency

Practical Issues

```
void buggy (void) {
    double a [1e8];
    int b [100];
    memset (b, 0, 10000);
    char *c = new char [100];
    memset (c, 0, 10000);
}
```

- Overhead per block
  - Release mode 16 bytes, debug 64 bytes
- Stack overflow
  - Too many local variables for default stack size or recursion too deep
- Stack corruption
  - Buffer overflow on local arrays
- Heap corruption
  - Block header wiped out or no man’s land is written to
- Heaps grab large pieces of memory from the OS
  - Since heaps are in user mode, they are quicker than asking the kernel
  - Allocation more efficient for small pieces (all kernel blocks rounded off to 4KB)
- When you run outside the heap into OS territory, hard crash on access violation

Practical Issues

```
DWORD *val, *shuf; // compiled in x64
main () {
    DWORD rnd = 3;  // LCG
    val = new DWORD [32];
    shuf = new DWORD [32];
    // generate random shuffle
    for (int j=0; j < 32; j++) {
          shuf[j] = rnd;
          rnd = (rnd * 5 + 11) & 0x1f;
    }
}

ThreadB () {
    for (int i = 0; i < 32; i++)
        printf (“%u\n”, val[shuf[i]]);
}
ThreadA () {
    memset (val, 0xff, 32*sizeof(val));
}
```

- Unless it’s extreme, heap corruption goes undetected
  - In debug mode, until the next new/delete operation sniffs something wrong and throws an assertion violation
  - In release mode, nothing happens until you crash
- Example: threadA corrupts the heap, threadB crashes
  - How to make these situations more suitable for debugging?
- Can ask the OS for the buffer using VirtualAlloc()
  - If writing outside page boundary, kernel does not tolerate any funny business, throws access violation immediately

Practical Issues

- Sometimes catching a crash obscures its cause

```
// SEH-style handler
__try {
    f(x);
}
__except ( MyCrashHandler ( GetExceptionCode() )) {
    // catch other exceptions here
}
```

```
f(x) {
  g(x);
}
```

```
g(x) {
  h(x);
}
```

- Writing a library that is used by someone else
  - Should you test their pointers for NULL?
  - Should you check if memory is valid using IsBadReadPtr, IsBadWritePtr, IsBadCodePtr, IsBadStringPtr?

```
MyLibraryAPI (char *ptr) {
    // how much checking to do
    // on validity of ptr?
}
```

Practical Issues

- One school of thought is to catch crashes, return explicit errors that help understand the problem
  - E.g., ReadFile returns error 998 (ERROR_NOACCESS)
- Another direction is to just crash without any checks
  - If someone is passing NULL or invalid handles, they're probably not checking for return codes; bugs should be made obvious to them

```
// homework #1 example
HANDLE pipe = CreateFile (pipename, ...);
while (true) {
    WriteFile (pipe, command, ...);
    ReadFile (pipe, buf, ...);
    // add rooms to queue, check uniqueness
}
```

- Meticulous return-code checking is important
  - Including WaitForSingleObject or ReleaseSemaphore